Brace your expr-essions

wiki.tcl-lang.org/page/Brace your expr-essions

See Also

double substitution
occurs when expressions are not braced

Static Syntax Analysis
helps to find unintentially-unbraced expressions

IEEE floating numbers
illustrates how unbraced expressions can be affected by $tcl_precision, along with an explanation by AMG

Description

KBK closed a follow-up to comp.lang.tcl with the following advice:

Also, it's important always, always to brace your expressions. Don't do

```
#not good!
set a [expr $b / $c]
```

but rather

```
set a [expr {$b / $c}]
```

Otherwise, your variables may undergo unexpected conversions numeric -> string -> numeric, you can lose precision, your expressions will be much slower, and you can even have security problems:

```
# don't run this code!
set x {[exec format C:\\]}
# the following line would execute format c:\
# set y [expr $x + 3]
```

PYK 2015-04-11: The only case where braces don't improve performance or enhance security is when a single literal value containing no substitutions or escape sequences is passed to expr. In this case, the literal value is passed straight through expr unmolested, and is tagged as a literal, making it eligible for byte-compilation. In the following example, braces would accomplish nothing:

```
set x [expr (2<<16)-1]
```

AMG: By "single literal value", PYK means exactly one argument containing no substitutions. expr (2<<16)-1 is fine, but expr (2 << 16) - 1 incurs a performance penalty.

AMG: The security problems of unbraced expressions are very similar to SQL injection attacks. Notice how sqlite's Tcl binding does its own variable expansion to avoid this very problem. Many, many sh scripts have this problem as well because the default is to apply multiple passes of interpretation.

Performance

AMG: Even in this bold era of bytecoding, bracing expressions is critical for performance. Check this out:

```
% tcl::unsupported::disassemble script {expr {2 + 2}}
ByteCode 0x02845A10, refCt 1, epoch 5, interp 0x02725500 (epoch 5)
  Source "expr {2 + 2}"
  Cmds 1, src 12, inst 3, litObjs 1, aux 0, stkDepth 1, code/src 0.00
  Commands 1:
      1: pc 0-1, src 0-11
  Command 1: "expr {2 + 2}"
    (0) push1 0         # "4"
    (2) done

% tcl::unsupported::disassemble script {expr 2 + 2}
ByteCode 0x02846D10, refCt 1, epoch 5, interp 0x02725500 (epoch 5)
  Source "expr 2 + 2"
  Cmds 1, src 10, inst 14, litObjs 3, aux 0, stkDepth 5, code/src 0.00
  Commands 1:
      1: pc 0-12, src 0-9
  Command 1: "expr 2 + 2"
    (0) push1 0         # "2"
    (2) push1 1         # " "
    (4) push1 2         # "+"
    (6) push1 1         # " "
    (8) push1 0         # "2"
    (10) concat1 5
    (12) exprStk
    (13) done
```

aricb 2005-12-10: As a bonus, braced expressions are generally *much* faster than their non-braced counterparts. Here's an attempt to illustrate the difference:

```
proc time_braces {} {

    set expressions {
        {1 + 1}
        {16 - 7}
        {(16 - 7)}
        {7 * 12}
        {81 / 3}
        {81 / 5}
        {16224308 + 123580329}
        {6.3 + 1.2}
        {6.3 - 1.2}
        {6.3 * 1.2}
        {6.3 / 1.2}
        {log(765) * 3}
        {(log(765) * 3) / 8.23}
    }

    puts [format " %-22s  %-8s  %-8s  %-8s  %-5s" "Expression" "-braces" "+braces"
"ratio" "% speedup"]
    puts " [string repeat - 22]  [string repeat - 8]  [string repeat - 8]  [string
repeat - 8]  [string repeat - 5]"

    foreach expression $expressions {
        set time1 [lindex [time "expr $expression" 100000] 0]
        set time2 [lindex [time [list expr $expression] 100000] 0]
        set ratio [expr {$time1 / $time2}]
        set percent [expr {100 * (1 - ($time2 / $time1))}]
        puts [format " %-22s  %8.6f  %8.6f  %6.3f:1  %5.2f" $expression $time1
$time2 $ratio $percent]
    }

    puts "\n -braces and +braces are in microseconds per iteration"
}
```

Results:

```
Expression              -braces   +braces   ratio     % speedup
----------------------  --------  --------  --------  -----
1 + 1                   2.334690  0.366480   6.371:1  84.30
16 - 7                  2.388280  0.359960   6.635:1  84.93
(16 - 7)                3.748160  0.356120  10.525:1  90.50
7 * 12                  2.438530  0.368200   6.623:1  84.90
81 / 3                  2.456060  0.395680   6.207:1  83.89
81 / 5                  2.518490  0.397380   6.338:1  84.22
16224308 + 123580329    2.726690  0.369530   7.379:1  86.45
6.3 + 1.2               3.237970  0.497100   6.514:1  84.65
6.3 - 1.2               3.199230  0.486350   6.578:1  84.80
6.3 * 1.2               3.279100  0.480630   6.823:1  85.34
6.3 / 1.2               3.257540  0.494340   6.590:1  84.82
log(765) * 3            6.202120  0.975590   6.357:1  84.27
(log(765) * 3) / 8.23   7.951270  1.142210   6.961:1  85.63

 -braces and +braces are in microseconds per iteration
```

RS 2007-01-09: These results puzzle me: the expressions contain no variable references, all constants. I thought the time advantage of braced expressions was because they could retrieve the numeric value of variables directly, without having to parse string reps - but in the above cases, the strings would have to be parsed either way...

AM: The numeric value of the strings is cached in the data structure that expr builds up during the parsing. So there is no real difference between numeric constants and variables with numeric values

RS: Well, but the only difference braces make, in my opinion, is that $, \, or [...] substitutions don't take place. If there are none,

```
expr 1+1
```

should be totally equivalent (in the eyes of expr) to

```
expr {1+1}
```

LV: It should be simple enough to enhance the above benchmark to include some variable uses to compare numeric strings vs numeric variables. - RS Sure, but my point was: how can "bracing" of constant strings lead to factors 0f 6..10?

LV Ask JH or AK for certain, but I suspect that the bracing puts the strings into a cache, with a Tcl object allocated for it, so that it doesn't need to be reparsed.

NEM: I think it is the grouping that makes the difference. With braces, expr gets a single argument and so can cache the parsed expression in that Tcl_Obj. Without the braces, it would have no Tcl_Obj available that represents the entire expression, so no good place to store the cache. This is a guess, but is backed up by augmenting the benchmark to also display the times for quote-grouped expressions. Add the following line to the benchmark:

```
set time3 [lindex [time "expr \"$expression\"" 100000] 0]
```

and adjusting the output to display this too, we get:

```
Expression              -braces   +braces   +quotes   ratio     % speedup
----------------------  --------  --------  --------  --------  -----
1 + 1                   3.290665  0.341749  0.340983   9.629:1  89.61
16 - 7                  3.392816  0.337900  0.340074  10.041:1  90.04
(16 - 7)                3.486216  0.342534  0.337474  10.178:1  90.17
7 * 12                  3.315694  0.339483  0.331504   9.767:1  89.76
81 / 3                  3.370662  0.357784  0.347305   9.421:1  89.39
81 / 5                  3.376001  0.354991  0.351884   9.510:1  89.48
16224308 + 123580329    4.481029  0.341851  0.339320  13.108:1  92.37
6.3 + 1.2               3.446636  0.349055  0.354368   9.874:1  89.87
6.3 - 1.2               3.511427  0.351007  0.346844  10.004:1  90.00
6.3 * 1.2               3.420894  0.348808  0.343441   9.807:1  89.80
6.3 / 1.2               3.471898  0.353067  0.358395   9.834:1  89.83
log(765) * 3            6.598790  0.759914  0.756337   8.684:1  88.48
(log(765) * 3) / 8.23   7.775013  0.829403  0.841776   9.374:1  89.33
```

So clearly the speed is mainly due to the grouping rather than braces avoiding substitition. (I also changed the braces line to *time "expr {$expression}"* to avoid any possible effects of pure-list optimisations, but this made negligible impact).

AMG: This is indeed the case, as shown by **tclCmdAH.C**.

```
743 Tcl_ExprObjCmd(dummy, interp, objc, objv)
748 {
758     objPtr = Tcl_ConcatObj(objc-1, objv+1);
759     Tcl_IncrRefCount(objPtr);
760     result = Tcl_ExprObj(interp, objPtr, &resultPtr);
761     Tcl_DecrRefCount(objPtr);
768     return result;
769 }
```

If there's only one argument to expr, then Tcl_ConcatObj() can return that object without having to make a temporary that dies on line 761. Tcl_ExprObj() will "shimmer" the object to an expression type. If this conversion is done on the object passed by the caller (instead of on a temporary object), it will be cached inside the caller's Tcl_Obj for the next time it is passed to expr.

RS: In other words, "if you don't want to brace, don't space" :^)

```
expr 1+1
```

would be faster than (the normally preferable style)

```
expr 1 + 1
```

But of course, most meaningful calls to expr involve at least one variable reference, so the advantage of bracing isn't really challenged.

slebetman: Indeed. I did the following modification to the test:

```
proc time_braces {} {
    set x 55
    set y 1.5

    set expressions {
        {1 + 1}
        {16 - 7}
        {(16 - 7)}
        {7 * 12}
        {81 / 3}
        {81 / 5}
        {16224308 + 123580329}
        {$x + $y}
        {$x - $y}
        {$x * $y}
        {$x / $y}
        {log($x) * $y}
        {(log($x) * $y) / 8.23}
    }

    puts [format " %-22s  %-8s  %-8s  %-8s  %-8s  %-5s" "Expression" "-braces"
"+braces" "quotes" "ratio" "% speedup"]
    puts " [string repeat - 22] [string repeat - 8] [string repeat - 8]  [string
repeat - 8]  [string repeat - 8]  [string repeat - 5]"

    foreach expression $expressions {
        set time1 [lindex [time "expr $expression" 100000] 0]
        set time2 [lindex [time [list expr $expression] 100000] 0]
        set time3 [lindex [time "expr \"$expression\"" 100000] 0]
        set ratio [expr {$time1 / $time2}]
        set percent [expr {100 * (1 - ($time2 / $time1))}]
        puts [format " %-22s  %8.6f  %8.6f  %8.6f  %6.3f:1  %5.2f" $expression
$time1 $time2 $time3 $ratio $percent]
    }

    puts "\n -braces, +braces and quotes are in microseconds per iteration"
}
```

and got the following result:

```
Expression               -braces   +braces   quotes    ratio     % speedup
----------------------- --------  --------  --------  --------  -----
1 + 1                    2.883460  0.344300  0.330900   8.375:1  88.06
16 - 7                   3.158980  0.353570  0.358150   8.935:1  88.81
(16 - 7)                 5.992560  0.350260  0.352260  17.109:1  94.16
7 * 12                   3.099940  0.349550  0.364370   8.868:1  88.72
81 / 3                   3.163450  0.369680  0.384460   8.557:1  88.31
81 / 5                   3.186310  0.407560  0.394200   7.818:1  87.21
16224308 + 123580329     3.966440  0.343470  0.345680  11.548:1  91.34
$x + $y                  4.658480  0.511210  4.512600   9.113:1  89.03
$x - $y                  4.634080  0.507070  4.548150   9.139:1  89.06
$x * $y                  4.634230  0.507840  4.615830   9.125:1  89.04
$x / $y                  4.706090  0.543070  4.610930   8.666:1  88.46
log($x) * $y             6.318600  0.700440  5.936520   9.021:1  88.91
(log($x) * $y) / 8.23    9.505550  0.856280 10.454150  11.101:1  90.99
```

```
-braces, +braces and quotes are in microseconds per iteration So the lesson is
unless you're doing something trivial like `(123*1024)`, always [brace your
expr-essions]!  The second lesson is be careful how you write your benchmark,
lest you accidentally test for something other than what you intended to test.
```

The Art of Unbraced Expressions

There is a time and a place not to brace expressions. Here's one example:

```
set a [dict create 1 one 2 two 3 three]
puts [expr max([join [dict keys $a] ,])]
```

The example above only works if the keys to the dictionary are numeric and there is a potential security vulnerability if the dictionary keys come from an uncontrolled source:

```
#danger, Will Robinson!
set a [dict create 1 one 2 two 3 three {[evil command here]} four]
puts [expr max([join [dict keys $a] ,])]
```

It's also posible to use the math function directly, without going through expr:

```
puts [::tcl::mathfunc::max {*}[dict keys $a]]
```

RS: In the following cases braced expressions don't work:

> operator in variable (which in most languages is impossible, but Tcl can do):

```
set op +
expr $x $op $y
```

PYK 2012-11: But the following works, and minimizes the drawbacks:

```
set op +
expr {$x} $op {$y}
```

AMG: But watch out for performance! See my tcl::unsupported::disassemble demonstration farther down this page.

> (parts of) expression generated by string manipulation:

```
expr [join $intlist +]
```

> GM function name is specified in a variable

```
set op sin
expr ${op}($x)
```

PYK 2012-11-28: but the variable could (should?) still be braced:

```
set op sin
expr ${op}( {$x} )
```

MJ: In 8.5 this can be done faster and safer like:

```
set op ::tcl::mathop::+
$op $x $y
```

and

```
$op {*}$intlist
```

In response to question on 2007-06-27 GWM Why does bracketing this not work:

```
set sign -
set valu 1.234
expr {$sign$valu}
```

when these do?

```
expr $sign$valu
expr {-$valu}
```

Answer: [expr] expects each variable to be a separate term in the expression. It does not concatente multiple varialbes into a single word the way Tcl does. However, normal tcl substitutions happen to variables in double quotes:

```
expr {"$sign$valu"}
expr {"-$valu"}
```

GWM this also works:

```
expr {[subst $sign$valu] + 1}
```

Lars H: It is because of the role of variable substitution in the expr little language -- a $varname is always an operand, never an operation. Having it this way is essential for byte-compilation to work -- consider what happens to

```
expr {2 + $a * [someProc]}
```

if substitution on $a is allowed to insert things like

```
(-2) ? 0 : 3
```

into the expression.

GWM: I think the very short answer is that the bytecode compiler will need to compile the expression differently if the operator is changed, so operators cannot be in variables unless you disable the compiler (by omitting the {}).

PYK: 2012-11-28 thinks the previous statement is wrong.

GWM: I think confusion arises as numerical values can be braced; anything else cannot.

```
set x 1.2
set y .3
expr {$x+$y}
```

is ok provided x & y are numbers. If

```
set x 1+.2
set y .3
```

then the above expr doesn't compile although expr $x+$y expands to expr 1+.2+.3 (= 1.5) which appears reasonable but $x is not a number.

PYK 2012-11-28: the previous example works fine form me in 8.5.13

Another example (extracted from Ask, and it shall be given # 8:

```
proc sqr {a} {return [expr {sqrt($a)}]}
proc tn {a} { return "2*tan([sqr $a])"}
sqr 8 ;# 2.8284271247461903
tn 2 ;# 2*tan(1.4142135623730951)
expr {[[sqr 8]]} ;# 2.8284271247461903
tn 2 ;# 2*tan(1.4142135623730951)

#However:
expr [tn 2] ;# 12.668238334084391
expr {[tn 2]} ;# 2*tan(1.4142135623730951)
```

With thanks to Lars for discussions. I have been playing with computer algebra for matrices, where the above problems occurred when creating Euler rotation matrices Euler Angles. My solution elsewhere has been to create dynamically a proc, something like:

```
proc myexpr {} [subst "expr {$x+$y}"]
```

which generates a compilable statement.

AMG: If you want expr to interpret its variables as math expressions rather than literal numerics, ask expr to recursively invoke itself (no braces around arguments to the inner expr:

```
set x 1+.2
set y .3
expr {[expr $x]*[expr $y]}
```

This gives you correct order of operations (replace the third line with expr $x*$y to see the problem), and you can be selective about which variables are expressions and which are numerics.

Also, check this out:

```
set x 1+.2
expr $x      ;# returns 1.2
expr {$x}    ;# returns 1+.2
```

The argument to the first expr is 1+.2, and the argument to the second expr is $x. In both cases it evaluates the expression, except that these two expressions are as different as night and day. In the second case, the expression evaluates to a non-numeric string. Unlike you and me, expr does not look to see whether the value of $x is a well-formed math expression.

KPV: Braced expressions also don't work when you do negation in the lazy way:

```
set n1 -3
set n2 -$n1
expr {2 + $n2} ;# error
expr 2 + $n2   ;# ok
```